Challenges and Opportunities for Dataflow Processing on Exascale Computers

Justin M. Wozniak,*† Michael Wilde,*† Ian T. Foster*†

* Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA
† Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA

***Abstract*—Exascale computers will produce a vast array of new resources to scientific software developers. In addition to the expected gains in core-count concurrency, exascale systems may feature multiple memories with various characteristics, a more complex storage hierarchy, and user control of power and resilience. These features will pose crippling software development complexities if suitable programming models are not developed. The dataflow model has the potential to greatly ease the programming challenges at exascale by implicitly determining concurrency, automating resource selection, expressing data transfer, handling faults, etc. In this work, we present ongoing efforts in the Swift/T language and runtime to support emerging exascale features.**

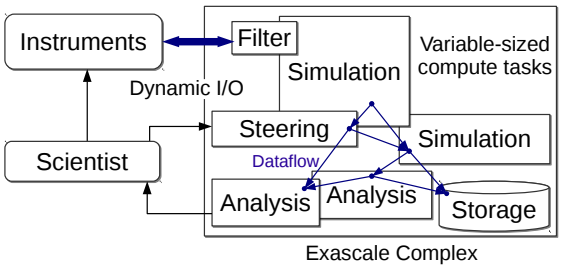

Fig. 1. Generalized exascale workflow: Coupled simulation and analysis tasks with large data transfer and dataflow control.

I. Introduction

Computational applications critical to society in areas such as materials design, climate modeling, and energy production and distribution must be developed quickly and correctly. Such studies are typically done via *computational experiments*, in which large numbers of simulation and analysis tasks are strung together into a workflow, formally or informally. Many innovative programming models to handle workflow specifications are based on an implicitly parallel dataflow language to support this model. Here, we posit that the dataflow model has the added benefit that it can exploit features of exascale computers, expected in ∼2023 [1].

Workflows and other outermost patterns such as parameter sweeps, searches, and optimizations can easily be expressed with dataflow languages. In this model, either statically or at runtime a dataflow structure is available to the runtime system, presenting the opportunity for many types of automated decisions for scheduling and resource management. Workflow applications on these systems will also introduce new requirements, such as high-performance data movement methods for in situ data analysis, and new challenges, such as varying reliability characterstics.

In this paper, we describe three key exascale feature areas that will be exposed to users at the application level. First, a more complex storage hierarchy is expected. New cache types such as scratchpad memory may be available, and heterogeneous RAM systems may have differing performance and reliability characteristics. Node-local storage may be available. These systems are expected to be available to the application or middleware via advanced operating system and runtime features. Second, tighter power budgets and programmer-controlled power scaling will likely be available. These will require the application level to make decisions about performance/power tradeoffs. Third, a more complex system will have more complex fault modes, many of which will allow the application to recover. The dataflow model is implicitly fault tolerant and easy to checkpoint; this integration will bring these features to exascale workflows and long-term campaigns.

Dataflow processing thus targets the most critical problems on the near-exascale and exascale roadmaps, such as the programming challenge posed by very large numbers (millions) of user threads per node, managing in situ communication, and heterogeneous components with respect to performance, power, and reliability. Dynamic performance behavior due to power and faults necessitate a naturally asynchronous programming model. Systems such as Argo [2], [3] offer rich low-level models and APIs to manage threads and tasklets, and DMEM [4] is investigating low-level data movement operations. What is needed is a very high level programming model to enable the rapid construction of applications that can make good use of these systems, while allowing developers to inject additional programming complexity where necessary [5]. Our most general expression of this model is shown in Figure 1, wherein multiple simulations and analysis jobs are coupled with a dataflow programming model that enables data-dependent processing and data transfer, including data transfer in and out of the exascale complex.

The remainder of this work is organized as follows. In Section II, we discuss workflows at exascale. In Section III, we describe the dataflow model as a programming model for exascale. In Section IV, we provide some systems challenges expected at exascale and show how dataflow is a viable solution. In Section V, we summarize our position.

II. Workflows at exascale

Scientific workflows are coarse-grained specifications linking software components and data definitions into an overarching structure in pursuit of a high-level application goal. The traditional way to express these workflows is to use a high-level scripting language to express the complete experimental structure, including data definitions, computation to be performed, and data dependencies, for high-performance execution. As shown in Figure 1, the experiment consists of many possibly multinode simulations, with dynamic tasks with varying properties and requirements performing auxiliary tasks, with possible connection to external instrument sensors or human users.

As an example, the Swift/T system revolutionized high-performance workflows by supporting fully in-memory workflows that combine in-memory libraries and data into a composite application capable of representing a complete computational experiment in a lightweight script. A key part of this effort involved scaling the workflow enactment system into a fully parallel runtime [6], [7] as well as developing compiler optimizations for distributed dataflow processing [8]. The system can thus maintain trillions of tasks, executing billions per second across large machines such as the Blue Gene/Q or Blue Waters.

Workflows at exascale may also be controlled by complex algorithms such as advanced sampling methods or machine learning. Swift/T-related efforts have integrated stateful Python and R tasks into the workflow to enable users to drop in evolutionary algorithms [9], [10] or machine learning toolkits [11] in order to parameterize simulations and carry out model exploration, parameter fitting, and classification. Numerical teams are recognizing the importance of ensemble studies and uncertainty quantification at exascale [12] and the complexity of the software changes that are required [13]. Additionally, concurrent-point optimization methods will become important in simulation-driven design [12]. The application of coarse-grained programming models and frameworks will speed developments in these areas.

III. Compositional programming models

Task-parallel programming models allow existing code (libraries or programs) to be rapidly developed into scalable applications. However, they generally do not capture the high-level *workflow structure* of the overall application. Concepts such as iteration, recursion, and reduction are lost if the user must coordinate tasks with the task-parallel library. It is difficult to compactly express these abstractions in the event-handling style required by the master-worker model. Additionally, data management is lost, and data dependencies must be encoded in an ad hoc manner.

The Swift/T language implementation represents a unique approach to this problem. Swift transparently *generates* an task-parallel ADLB program [14] from a high-level script, which contains data definitions, data dependencies, and links to external native code (i.e., C/C++/Fortran). This program can then be run on an MPI-based high-performance computer. Swift/T also has many other practical features to enable dataflow processing at scale [15].

Other high-performance languages are beginning to investigate more dynamic, irregular patterns. Chapel, for example, supports low-level tasks and data dependencies [16]. Charm++ supports dataflow programming through Structured Dagger [17]. OpenMP 4.0 (c. 2013) [18] includes coarse-grained task dependency features. The variety of these models is great, but all are recognizing the need for higher-level constructs that encapsulate bigger fragments of computation. By focusing on this as a first-class problem, we can engage with these development teams and make synergistic progress on this computer science problem.

Bringing such systems to exascale will involve adaptations, improvements, and accommodations for the challenges expected at exascale, such as the trend toward full-features node operating systems; the explosion in computational concurrency on each shared-memory node and properties of heterogeneous processors, memories, and storage systems; the need for efficient in situ processing; and fault tolerance and checkpointing issues. The high-performance computing community faces a challenge in solving the problem of effectively programming such complex patterns given the novel hardware expected on such machines.

Hierarchical programming is a natural fit for these exascale systems challenges. Composing disparate software elements and addressing the multilanguage problem is a key part of the workflow problem. Since Swift is a hierarchical language, all Swift applications use at least two languages; Swift and the task-specific language. Our technique involves wrapping existing codes with bindings for higher level languages, then building composite applications that stitch together the functionality into a complete experiment. The resultant hierarchical execution model provides a *control system* –like model capable of adapting to dynamically changing resources and requirements. The application program can thus be divided into a logical *plant* containing low-level, performance-critical user tasks, and *control* functionality in the Swift level. The control level is capable of optimizing the resource usage by tasks, including, e.g., task restart (Swift itself would be run on the most reliable but not necessarily fastest resources).

A high-quality control level would require multiple features. It would need to be able to build up the dataflow graph and perform critical path analysis. Doing so requires some estimate of task runtime length. It would benefit from other hints, such as task resource requirements, reliability requirements, restartability, etc. It would also require data about the underlying system, such as the characteristics of the compute capabilities, storage systems, network topology, etc. While an optimal solution to the resource mapping problem is known to be NP-hard, we think many problems can be solved well enough by performing critical path analysis on a simplified task runtime model, maximizing the performance of those tasks, and filling in the rest in an ad hoc manner.

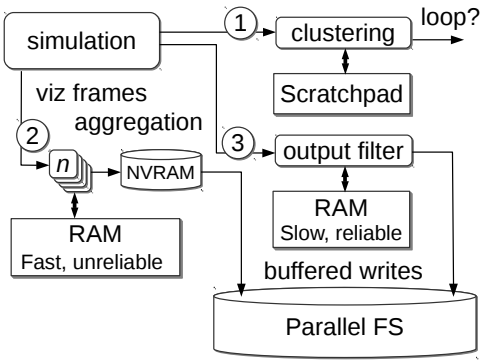

Fig. 2. Complex memories for in situ processing.

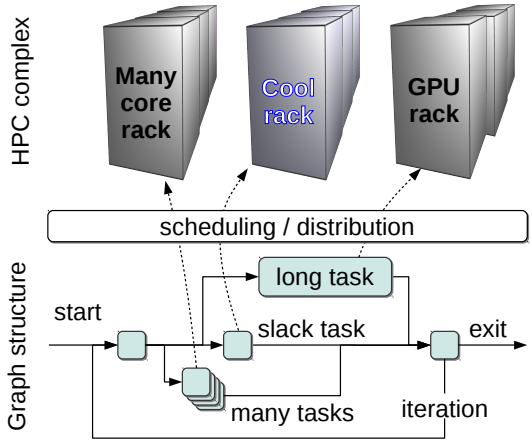

Fig. 3. Power-aware dataflow scheduling.

IV. Exploiting dataflow on exascale systems

Emerging high-end hardware and operating systems are expected to have many more features for storage, power management, and fault tolerance. These features will have to be controllable by the user program; under dataflow, this will be managed by the language runtime, which will be able to use these features because of the program structure exposed. This greatly eases the burden on the scientific programmer, who writes parallel task components, allowing the higher level system to manage emerging system features.

A. Complex storage architectures

Storage systems on exascale machines are expected to be more complex, with nonvolatile random access memory (NVRAM) at multiple possible locations to boost I/O [19]. If this is available to programmers, it will offer a useful medium for complex patterns that mingle RAM with NVRAM, intersperse complex checkpointing strategies, and enable users to program against data that may be managed by the runtime.

Consider the model workflow shown in Figure 2, in which in situ simulation data is collected by three data-dependent pipelines. Pipeline (1) performs k-Means clustering on some simulation output parameter, a method that could greatly benefit from scratchpads [20]. The runtime would thus allocate scratchpad memories to these tasks where possible and provision memory near to allocated processors. Other tasks will be allocated remaining RAM. Noncritical tasks, such as those that aggregate data for human visualization (2), or otherwise flagged as noncritical (e.g., tasks that are I/O-bound), could be given unreliable RAM allocations backed by NVRAM for capacity. An output filter that looks for interesting events (3) could be backed by slower but more reliable RAM so that statistics about such events are trustworthy.

In other patterns, data chunks could loaded from the filesystem, possibly directly from their object representations [21], possibly via collective operations to NVRAM [22]. User dataflow code could carry out an explicit checkpoint by creating a data dependency to disk. If this is the data item required by the subsequent task, progress will block on the checkpoint, although other tasks could continue to make progress. Dependencies on human input could also be specified.

High-quality dataflow runtimes capable of managing such workloads would have to support a broad range of storage policies. These would include explicit storage locations (RAM, NVRAM, local FS, global FS), as well as runtime-managed locations (LRU, etc.). These would enable a coarse-grained scratchpad memory system that could strike a good tradeoff between programmer control of all locality and total system control (as in MapReduce [23]).

B. Power management

Power is expected to be a major constraint in the design of exascale computers. Internode, intercabinet, and node-to-storage costs (for both latency and energy) will be varied and inter-dependent. Programmers will be confronted with complex considerations for task and data placement and scheduling to reduce communication volume, distance, and energy use. In the case of an ensemble application coordinated via dataflow, the problem is easier to address automatically.

The dataflow model also makes possible the automatic reduction of communication volume and distance, and the determination of when functional units can be quiesced to save energy. As shown in Figure 3, the data dependency graph and task encapsulation expressed in hierarchical dataflow provide rich information about requirements. Based on their place in the critical path model, tasks can be allocated to appropriate processor types running in appropriate power modes. Data storage enhancements discussed previously are also relevant here, because algorithms for placing tasks and data in more optimal proximity with respect to power would reduce the energy cost of data movement.

C. Fault tolerance, resilience, and elasticity

In the dataflow model, fault tolerance and elasticity are connected. Software fault tolerance will be required on emerging exascale machines. Elasticity is desirable for in situ analysis workflows in which the amount of work for analysis tasks varies unpredictably over time. In both cases, the number of nodes available to the overall application will change as

partitions are added, are lost, or change size. The dataflow model allows the runtime system to maintain application state and migrate work to new task executors (workers), given information about faults from emerging OS and/or MPI features [24].

The first challenge is to allow the dataflow program to gain or lose workers. This is relatively easy since the data dependency specification on each task allows task restart (while practical user tasks may have side effects, these are outside the dataflow specification). The second challenge is to allow the dataflow processing engine to be just as resilient and elastic. This work will result in novel computer science contributions. Additionally, high-level user-managed fault response through language extensions such as exception handling and/or optional values [25], [26] could be viable. This challenging area would determine the implications of these models in a distributed-memory language.

Many optimization methodologies require underlying techniques that are available through low-level programming but are not yet provided by higher-level dataflow programming. Some examples are motivated by the branch-and-bound method, which solves an integer optimization problem by solving the "relaxed" continuous problem and updates bounds on the optimal value and relevant search region. Individual branches (tasks) pursuing a solution in one region may be affected, or even eliminated, by updates to other regions. This is difficult to implement in the dataflow model with no task side effects. However, many applications have such patterns, and a high-productivity model that mingles dataflow with the capability to implement these patterns is highly desirable.

V. Summary

In this paper we have outlined multiple potential applications of dataflow-based programming and runtime systems to the challenges of exascale computing. We described how a coarse-grained dataflow abstraction can capture the overarching patterns in exascale workflows, particularly in describing data-dependent analysis tasks attached to large-scale simulation ensembles. This in situ data transmission and analysis model enables a highly productive programming model, where components can be easily coupled and decoupled for varying computational experiments. We also described how this programming model allows the runtime system to automatically make computation and data placement decisions relevant to multiple exascale systems challenges.

Our position is that exascale computing offers an exciting opportunity for dataflow processing to solve challenges conventionally addressed by systems designers and implementors. The Swift/T system is but one attempt to address these problems, and we invite discussion at the DFM workshop about this research and development area.

References

[1] T. Trader, "Paul Messina shares deep dive into US exascale roadmap," June 2016, in HPCwire, https://www.hpcwire.com/2016/06/14/us-carves-path-capable-exascale-computing.

[2] H. Lu, S. Seo, and P. Balaji, "MPI+ULT: Overlapping communication and computation with user-level threads," in *Proc. HPCC*, 2015.

[3] S. Perarnau, R. Thakur, K. Iskra, K. Raffenetti, F. Cappello, R. Gupta, P. Beckman, M. Snir, H. Hoffmann, M. Schulz, and B. Rountree, "Distributed monitoring and management of exascale systems in the Argo project," in *Distributed Applications and Interoperable Systems*, ser. Lecture Notes in Computer Science, A. Bessani and S. Bouchenak, Eds. Springer International Publishing, 2015, vol. 9038, pp. 173–178.

[4] A. J. Peña and P. Balaji, "Toward the efficient use of multiple explicitly managed memory subsystems," in *Cluster*, 2014.

[5] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, 2015.

[6] J. M. Wozniak, T. G. Armstrong, M. Wilde, K. Maheshwari, D. S. Katz, M. Ripeanu, E. L. Lusk, and I. T. Foster, "Turbine: A distributed-memory dataflow engine for extreme-scale many-task applications," in *Proc. Workshop on Scalable Workflow Enactment Engines and Technologies*, 2012.

[7] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster, "Turbine: A distributed-memory dataflow engine for high performance many-task applications," *Fundamenta Informaticae*, vol. 28, no. 3, 2013.

[8] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster, "Compiler techniques for massively scalable implicit task parallelism," in *Proc. SC*, 2014.

[9] J. M. Wozniak, T. G. Armstrong, K. C. Maheshwari, D. S. Katz, M. Wilde, and I. T. Foster, "Interlanguage parallel scripting for distributed-memory scientific computing," in *WORKS '15: Proceedings of the 10th Workshop in Support of Large-Scale Science*, 2015.

[10] J. Ozik, N. Collier, and J. M. Wozniak, "Many resident task computing in support of dynamic ensemble computations," in *Proc. MTAGS at SC*, 2015.

[11] ——, "From desktop to large-scale model exploration with Swift/T," in *Proc. Winter Simulation Conference*, 2016.

[12] DOE ASCR Exascale Mathematics Working Group, "Applied mathematics research for exascale computing," 2014.

[13] E. Phipps, H. C. Edwards, and J. Hu, "Realizing exascale performance for uncertainty quantification," in *DOE Workshop on Applied Mathematics Research for Exascale Computing*, 2013.

[14] E. L. Lusk, S. C. Pieper, and R. M. Butler, "More scalability, less pain: A simple programming model and its implementation for extreme computing," *SciDAC Review*, vol. 17, 2010.

[15] J. M. Wozniak, M. Wilde, and I. T. Foster, "Language features for scalable distributed-memory dataflow computing," in *Proc. Data-Flow Execution Models for Extreme-Scale Computing at PACT*, 2014.

[16] Cray Inc., "Chapel language specification version 0.96," available at http://chapel.cray.com.

[17] L. V. Kalé and M. A. Bhandarkar, "Structured Dagger: A coordination language for message-driven programming," in *Proc. Euro-Par'96 Parallel Processing*, 1996.

[18] "OpenMP 4.0 complete specifications," available at http://openmp.org.

[19] S. Ahern, "Scientific discovery at the exascale: Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization," 2012.

[20] M. A. Bender, J. Berry, S. D. Hammond, B. Moore, B. Moseley, and C. A. Phillips, "k-means clustering on two-level memory systems," in *Proc. MEMSYS*, 2015.

[21] C. Karakoyunlu, D. Kimpe, P. Carns, K. Harms, R. Ross, and L. Ward, "Towards a unified object storage foundation for scalable storage systems," in *Proc. IASDS at Cluster*, 2013.

[22] J. M. Wozniak, H. Sharma, T. G. Armstrong, M. Wilde, J. D. Almer, and I. Foster, "Big data staging with MPI-IO for interactive X-ray science," in *Proc. Big Data Computing*, 2014.

[23] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings on Operating Systems Design and Implementation*, 2004.

[24] "BEACON and EXPOSE: Backplanes for the ARGO system," http://www.argo-osr.org/overview/backplane.

[25] "Data.Maybe reference," optional values in Haskell are described here: https://hackage.haskell.org/package/base-4.2.0.1/docs/Data-Maybe.html.

[26] "Swift programming guide," section on Optional Chaining. Note that this is the Apple product named Swift and is not related to the Swift language described in this paper.